Domain-Specific Languages and Digital Preservation Supporting Knowledge-Management

Tobias Dehling
Ali Sunyaev

Veröffentlicht in:
Multikonferenz Wirtschaftsinformatik 2012
Tagungsband der MKWI 2012
Hrsg.: Dirk Christian Mattfeld; Susanne Robra-Bissantz

Braunschweig: Institut für Wirtschaftsinformatik, 2012

Domain-Specific Languages and Digital Preservation Supporting Knowledge-Management

Tobias Dehling

University of Cologne, Faculty of Management, Economics and Social Sciences, Albertus-Magnus-Platz, D-50923 Cologne, E-Mail: dehling@wiso.uni-koeln.de

Ali Sunyaev

University of Cologne, Faculty of Management, Economics and Social Sciences, Albertus-Magnus-Platz, D-50923 Cologne, E-Mail: sunyaev@wiso.uni-koeln.de

Abstract

Domain-specific languages (DSLs) are languages most suitable for a specific application domain. They abandon generality to increase expressiveness and ease of use. DSLs are an useful alternative to general-purpose languages, but their appropriateness and profitableness should be carefully considered. In this paper the utilisation of DSL knowledge to improve digital preservation practices is elaborated, which leads to the following results: A DSL for emulator development reduces implementation effort and increases comprehensibility and durability. The DSL XML provides format standardisation for information stored in plain text. A DSL especially for the domain digital preservation is not practicable and digital pre-servation is better supported by the right mix of DSLs. The results are particularly interesting from a knowledge-management perspective since gathered knowledge should be preserved.

1 Introduction

Digital preservation “involves selecting, preserving and managing digital information in ways that promote easy discovery and retrieval for both current and future uses of that information” [4]. Keeping the data comprehensible is necessary since preserving incomprehensible data is pointless. Domain-specific languages (DSLs) could enhance comprehensibility because they increase expressiveness and ease of use [5]. However, it is still necessary to explore how knowledge about DSLs can be utilised for digital preservation.

An approach to leverage domain-specific knowledge in digital preservation could be beneficial because domain-specific languages could bear some advantages. In software-engineering, for example, the use of DSLs leads to higher productivity, efficiency, and quality, supports end-user programming, preserves insight, and supports optimisations and transformations that would be infeasible with general-purpose languages (GPLs) [14]. Furthermore, improvements in digital preservation could be rewarding in knowledge- management since the gathered knowledge or

the systems providing the information could remain useful for a longer time span. Therefore, they need to be preserved and have to remain accessible; this is the objective of digital preservation. A comprehensible definition of knowledge-management proposed by Young is: “Knowledge Management is the discipline of enabling individuals, teams and entire organizations to collectively and systematically capture, store, create, share and apply knowledge, to better achieve their objectives.” [15] The knowledge-management definition underlines the relevance of digital preservation and the utility of improvements in digital preservation for knowledge-management because both disciplines have intersecting objectives: The selection, preservation and management of information/knowledge to promote easy discovery and retrieval for both future and current uses of information/knowledge.

The aim of this paper is to explore possibilities to apply DSL knowledge to digital preservation. The paper is structured as follows: In the chapter 'Domain-Specific Languages' we define and illustrate DSLs and oppose the advantages and disadvantages of using a DSL instead of a GPL. In the chapter 'DSL-Application Ideas for Digital Preservation' we propose some ideas how DSLs could be useful for digital preservation and analyse their benefits. Finally, we summarise and conclude the paper in the last chapter titled 'Conclusion'.

2 Domain-Specific Languages

2.1 Definition

DSLs provide notations and constructs most suitable for a specific application domain, which leads to substantial gains in expressiveness and ease of use in contrast to GPLs. These gains are achieved by abandoning generality to obtain expressiveness in a specific domain [5]. Due to the narrow focus of a DSL they are usually part of a larger system [8]. GPLs, on the other hand, provide a basic set of functionality which the users utilise and combine to implement their programs [2]. Such a general approach makes GPLs suitable for many different problems, but expressiveness is impeded by formal noise [13].

The following examples illustrate the rather formal description of DSLs. The OpenGL Shading Language (GLSL[1]) is a language that gives developers more control of the rendering pipeline. Developers can write vertex and fragment shaders to achieve better or different results than they could using OpenGLs fixed-function rendering pipeline. The GLSL is a DSL since it was developed to program the graphics processing unit (GPU) and it provides only the functionality necessary to program the GPU, like data-structures for two- to four-dimensional vectors and matrices and methods for the dot and cross product. The GLSL looks much like a typical programming language and a shader written in it looks like a program, but DSLs do not necessarily have to be programming languages. The Web Ontology Language (OWL[2]) is a language to represent knowledge in such form that it can be processed by machines. The OWL is a DSL designed for the domain knowledge representation. Documents written in OWL are human-readable and just slightly related to source files written in programming languages. The Hyper Text Markup Language (HTML) is a DSL, too. HTML is used for document markup. It provides markup tags which define how the resulting document, usually a web page, should look or provide meta information. Obviously, HTML has only a very distant relationship with

[1] For more information on GLSL see [7].
[2] See http://www.w3.org/TR/owl-features for more information on the OWL.

programming languages. Furthermore, DSLs do not have to be literal. The modern musical notation for example uses symbols to represent music in written form. It is a DSL with the domain musical notation and provides language constructs, like the G clef or a quarter note, and defines how those can be combined and interpreted.

2.2 Language Development

Wile proposes three DSL implementation approaches in [13], which explicate the complexity of DSL development. Language development necessarily requires the design of the language itself. A language is designed by defining its syntax and semantics. When the DSL is designed it can be implemented in various ways. It can be implemented as a new independent language, an extension of an existing language, or based on Common Off-The-Shelf (COTS) products.

To implement a new language one needs to use tools like LEX[3] and YACC[4]. With LEX the developer can create a lexical analyser that finds the tokens (words and symbols) in an input stream if they are a part of the defined language. When the input stream has been converted to a sequence of tokens, it can be used as input for a parser, which can be created with a tool like YACC. The parser checks the sequence of tokens for correct syntax and prepares the data for further processing. The proposed approach is one way of developing a tool that can read and provide the information written in an independently crafted language.

The language extension approach aims at hiding the complexity of the program from the end-user. The language is extended with new simple language constructs, which provide all the functionality commonly used in the domain. This empowers the end-users to solve their tasks by using the simple language extensions instead of writing complex code in the host language (the language that is extended). Therefore, the end-users only need rudimentary knowledge of the host language and have to know the implemented language extensions to work with the extended language in their domain.

Furthermore, the language could be created based on COTS products like Microsoft Access, Microsoft PowerPoint, or the Extensible Markup Language (XML). In order to create a language based on Access the developer has to design the relational database created with Access according to the syntax of the language. Such a relational database could be designed by creating a relation for each language construct where the relation's attributes describe the attributes of the language construct. To illustrate the usage of COTS products for language development, we propose the following example: In a fictitious DSL used to design organizational charts you could construct a relation 'POSITION', for the language construct 'position', with the attributes (ID, PARENT_ID, EMPLOYEE_ID, NAME, DESCRIPTION, ...). Attribute values can be restricted by SQL-statements so that the user is not able to enter invalid data like an 'EMPLOYEE_ID' that does not exist.

There is no 'best' approach to DSL implementation. If the decision to use a DSL is made one will have to choose a way of implementation that fits the characteristics of the DSL and the expertise available in the organization. More ways of DSL implementation exist, but the three approaches described above resemble the main categories.

[3] See http://dinosaur.compilertools.net/lex/index.html for more information on LEX.
[4] See http://dinosaur.compilertools.net/yacc/index.html for more information on YACC.

2.3 Advantages versus Disadvantages of DSL Application

In order to decide whether it is beneficial to use a DSL instead of a GPL it is necessary to oppose advantages and disadvantages. DSLs provide domain-specific notations, which are usually not available in GPLs. Domain-specific notations are an advantage since users will be more productive if they can work using accustomed domain-specific notations. A DSL also offers analysis, verification, optimisation, parallelisation, and transformation methods, which might be unfeasible or too complex to be implemented in a GPL. Furthermore, domain-specific constructs that can only be expressed indirectly and uncomfortably in a GPL can be better expressed in a DSL since a DSL is designed to do exactly this [5]. The DSL spares the user of having to deal with the notational noise that comes with general constructs of general-purpose languages. Being able to express things concisely leads to further advantages. More things can be read at once, which increases comprehensibility. Comprehensibility is increased because relations that might have been lost in formal noise, if a GPL had been used, can be made. Additionally, increased conciseness leads to easier writing, easier writing leads to fewer clerical mistakes, and both advance productivity. The productivity is further increased because errors are expressed in domain-specific terms since the language consists of domain-specific terms. Moreover, errors expressed in domain-specific terms are easier to understand. A great advantage is that domain experts are enabled to code programs and specifications on their own since development with DSLs requires mainly domain know-how instead of programming skills [13]. Another advantage is that DSLs do not have a lot of elements like a GPL so that the runtime efficiency is not compromised by interdependencies of different elements. The implementation of the language is also quite reliable because the small scope of a DSL makes the verification of its implementation easier in contrast to a GPL [8]. Another plus is that best practices can be incorporated in DSLs, so that the users apply them without having to learn them. Just the language developers are required to implement and update the DSL accordingly.

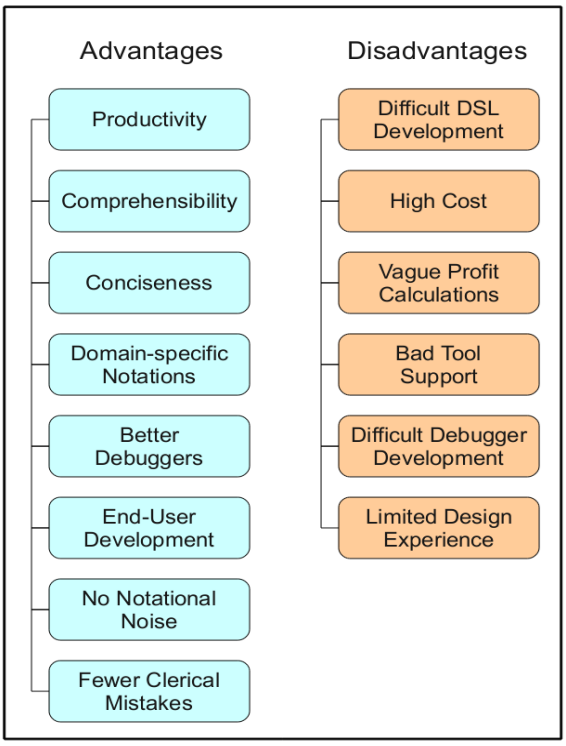

Figure 1: Advantages and disadvantages of DSLs

Disadvantages of using DSLs exist as well. DSLs are difficult to develop because language development expertise as well as expert domain knowledge is required. There are various techniques of DSL development, as described in chapter 'Language Development', which requires a thorough examination of factors influencing the achievement of the development goals to choose the right technique. Additionally, a large user community creates costly efforts for development of training material, language support, standardisation, and maintenance [5]. After all the effort for design and implementation, the DSL can still “be a bit artificial for the domain” [13]. Debugging simulations and error notification get more difficult because error messages have to be in the context of the domain instead of the programming language. There might be bad tool support for a self-designed DSL since a DSL that has been just created is unlikely to be supported by previously developed tools. Additionally, external restrictions, like support of special platforms, constrain the DSL development [13]. Furthermore, there is limited design experience for systems based on DSLs because DSLs are not as commonly used as GPLs. There are also not many enterprise resources available for DSL development since DSLs are usually part of a larger system and focused on a narrow usage domain, thus they are just entitled to access a small percentage of the resources available for the system they belong to [8].

Obviously, there are some good reasons for choosing to use a DSL instead of a GPL, but one should always consider what they want to achieve and what possibilities there are to do so. In order to write a static web-page almost everyone would choose to use HTML, which is well documented and widely used. On the other hand, you would not develop a DSL to implement some small functionality that you will only use once. Commonly used DSLs are usually a good choice since they bear a lot of the advantages but only a few of the disadvantages. Preexisting DSLs that are not as common are likely to generate more costs than commonly used DSLs because they require more training and might not do exactly what one expects. On the contrary, preexisting DSLs do not generate a lot of development costs. A self-developed DSL generates the most costs since language development is difficult, but they can also become exactly what you want them to be. DSL development is a good choice if the DSL is simple or expected to be used a lot and to solve a lot of problems. As a result, you can say that DSLs can be useful, but it should always be carefully considered whether the benefits are larger than the efforts for development and use.

3 DSL Application-Ideas for Digital Preservation

3.1 Durable Emulation Through a Domain-Specific Language

Emulation durability could be enhanced by introducing a DSL for emulator development. To justify this, two approaches proposed by Rothenberg in [6] will be described and then we will suggest how these approaches could be improved by a DSL for emulator development.

Rothenberg states that many preservation strategies require the preservation of bitstreams and assumes that the bitstreams can be preserved by repeatedly copying the bitstreams on new storage media. The copying will ensure that the bitstreams remain readable. He sees emulation as a strategy that creates a new program which runs on the current hardware generation and enables the computer to perform like a computer of a previous generation in order to run any program from that generation. Emulators can be implemented on future generations of hardware, if those provide the functionality of previous generations. However, in Rothenberg's

opinion the availability of that functionality is quite plausible since the capabilities of computers “are founded on simple, universal, mathematical, and logical operations” [6], which will remain useful independent of newly added capabilities.

His first approach is called 'Chained Emulation'. In Chained Emulation an emulator for $generation_1$ hardware that runs on $generation_2$ hardware is developed. When $generation_2$ hardware is likely to be antiquated an emulator for $generation_2$ hardware that runs on $generation_3$ hardware is developed. This method is carried on for every $generation_n$ for which an emulator that runs on $generation_{n+1}$ is developed. In order to run a $generation_1$ software to display a $generation_1$ record (data) on $generation_3$ hardware one would run an emulator for $generation_1$ hardware in an emulator for $generation_2$ hardware that runs on $generation_3$ hardware. This way software of every generation can be used to display records made in that generation by nesting emulators inside emulators.

The second approach is called 'Rehosted Emulation'. Rehosted Emulation is quite similar to Chained Emulation, but instead of nesting emulators inside emulators, emulators are rehosted on the current hardware generation. When hardware $generation_3$ becomes obsolete, an emulator for $generation_3$ hardware that runs on $generation_4$ hardware is developed, as it is done in Chained Emulation. However, in Rehosted Emulation the emulators for $generation_2$ and $generation_1$ hardware are additionally rehosted on $generation_4$ hardware. Rehosted Emulation is beneficial when the performance loss through emulator nesting is greater than the cost for rehosting emulators.

Developing those emulators is a demanding task since an emulator is not only required for every hardware generation, but as well for different computer platforms using different hardware generations. Furthermore, an emulator needs to work correctly. Otherwise, data might be lost if a future emulator needs a previous emulator to access old data. Since emulators are viewed as black boxes, there is not much that can be done if an old emulator, which is necessary for Chained Emulation, encounters an error. To facilitate emulator development Rothenberg mentions two techniques. The use of a virtual machine for running emulators and the development of emulators “in a single, standardized language that is well formalized and semantically rigorous” [6]. The second approach, the development in a single language, sounds like the use of a DSL for emulator development. In the remainder of the paragraph, the usefulness of a DSL for development of emulators for the emulation strategy will be elaborated. As mentioned earlier, the emulation strategy requires a lot of emulators. There is at least one emulator for every important platform available in hardware $generation_n$ to all important platforms in the previous hardware $generation_{n-1}$. With emulators developed in a DSL the effort would be reduced since only one emulator implementation and one implementation of the DSL would be necessary for every important platform of every hardware generation. Furthermore, it is not necessary to implement the DSL on every platform. If the DSL is correctly implemented on one platform of the current generation and emulators developed in the DSL can be executed on this platform, they should run with other implementations of the DSL as well. They should run with other implementations as well since the implementations only provide the functionality required by the DSL for a specific computer platform. A further advantage of using DSLs is that DSLs are human readable. Therefore, emulator specialists are more likely to fix an old emulator that stopped working for some reason, than if they have to understand source code in a GPL that might not be used any longer. The decreased implementation effort increases emulation durability because less required effort leads to less errors and leaves more time to focus on

software quality. Development in a single DSL for emulator development increases durability because there will never be emulators written in a 'forgotten' language, which would be hard to maintain and debug. Furthermore, developers will always be able to understand source code of already implemented emulators easily in order to fix errors, learn or reuse code. Best practices for common tasks encapsulated in the DSL itself improve quality and durability as well. Obviously, enhancement of the emulation strategy through a DSL is beneficial for knowledge-management if the emulation strategy is used for the preservation of information objects relevant for knowledge-management. Using the emulation strategy for digital preservation in knowledge-management is for example favorable when information objects / knowledge is not accessible otherwise, when it is possible that transformation will result in a loss of knowledge, or when transformation is too expensive.

Supposing Rothenberg is right in assuming that a future computer will provide “all the logical functions that the old computer performed” [6], it should be possible to maintain a DSL for emulator development. Such a DSL can be maintained since it has to support a growing amount of functionality but does not have to deprecate language constructs, which would prohibit execution of emulators using these constructs. Therefore, the durability of emulation can be enhanced through the application of a DSL for emulator development.

3.2 Facilitation of Format Standardisation with XML

Format standardisation is useful for digital preservation because standardized file formats provide good interoperability and interchangeability, there is good tool support for developers since standards are widely used, and standards are probably longer in use than non standard solutions. The DSL XML[5] is such a standard, which is continuously becoming more popular [3].

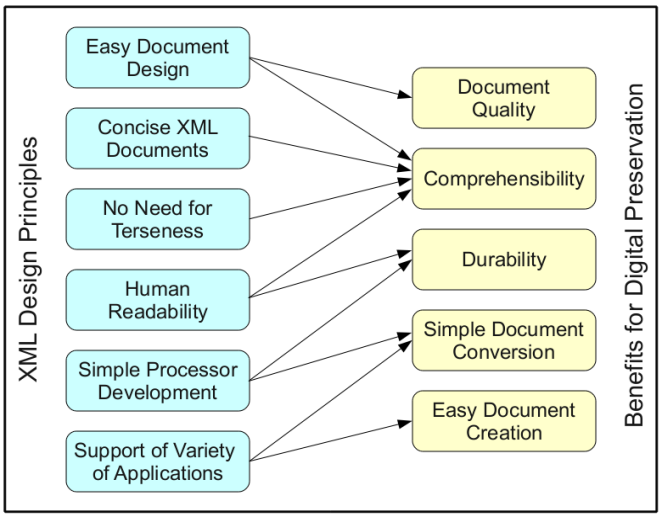

Figure 2: Relationship between XML design principles and benefits for digital preservation

XML is a markup language for creating documents that are human- and machine-legible. XML documents contain the information that should be stored in the document and additional markup tags that define the structure of the document. The tags are not predefined and have to be defined by the developer. Therefore, XML is a meta-language that can be used to define an arbitrary language for representing textual information [9]. XML seems to be useful for digital

[5] For a short introduction to XML see [9], p. 21-44.

preservation since the design goals of XML fit the needs of format standardisation and digital preservation (see figure 5). The relevant design principles are the support of a variety of applications, easy development of applications that process XML, easy design of XML documents, conciseness of XML documents, easy creation of XML documents, no need for terseness in XML documents, and XML documents should be human-legible and reasonably clear [11]. The support of a variety of applications aids digital preservation because applications for digital preservation that produce records to be preserved can store the records directly in XML and other formats can be easily converted to XML. The easy development of XML processors is useful because it makes more likely that XML processors can be generated on future hard- and software generations. The easy creation of XML documents is beneficial since plain text editors are more likely to be available in the future than some complex editing software. Furthermore, the easy creation shifts the focus of the archiving process, along with the easy design, more to the stored data than to the creation of the document itself. The design additionally aids in the understanding of the content because it is not necessary to comprehend some complex file format in order to extract the stored information. The fact that XML documents are human-legible further aids in the understanding since no software needs to be archived along with the record and the data can be evaluated directly in contrast to relying on the correct representation by some application. Since terseness is not required, meaningful tag names like for example 'accountnumber' can be used instead of abbreviations like 'anum'. The clearness and conciseness of XML support the understandability as well. The successful use of XML depends of course on human action. It is important that the quality of the created documents is controlled because documents heavily violating the design principles are not of much use. However, human action should be manageable and is assisted by XML Schema (XSD[6]). XSD is a DSL which is used to define a set of rules to which an XML document must conform [9]. XSD enables the developer to ensure at least some degree of document quality by defining the rules accordingly.

Since XML documents are plain text files, they are suitable to preserve information that can be stored as plain text. For other information like images XML could just be used to store some information how the image data is to be interpreted, but the main capability of XML concerning digital preservation is the format standardisation for plain text files. A Dutch research project [10] has tested the capability of XML to store different record types. The results are that XML is suitable for storing the context, content, structure, and behaviour of text documents. In combination with a stylesheet[7], XML is also able to reproduce the appearance. Spreadsheets could also be represented smoothly with XML. Since the format of e-mails is standardized, they have a sender, recipient, subject, content, etc., e-mails can be easily represented with XML; it is just necessary to define matching XML tags. XML was also found suitable to represent whole database systems[8]. In order to be able to represent the appearance of the user applications it was additionally necessary to store the technical and functional documentation of the database. These results back the implications derived from the XML design principles and show that XML is useful for preserving information that can be represented with plain text. Knowledge-Management can benefit from the enhancement of digital preservation through format standardisation with XML: The ability to concisely capture, store, preserve, and easily share digital information facilitates the activities of knowledge-management.

[6] For a short introduction to XSD see [9], p. 57-69.
[7] A stylesheet is an additional document that defines how a XML document should be displayed.
[8] In this case whole database system means the database, the database management system and user applications.

3.3 A Domain-Specific Language for Digital Preservation

A DSL tailored to the domain digital preservation, which could diminish the problems arising in digital preservation, would be advantageous. The idea assessed in this chapter is the development of a DSL especially for the domain digital preservation. Obviously, DSLs cannot be used to vanquish the physical deterioration of storage media, but there might be a DSL that makes digital preservation easier. Unfortunately, it is quite difficult to devise the design and task of such a language.

The problem is that information objects that are valuable enough for preservation are created by companies in many different industries, by different governments, by the scientific sector, by individuals, etc.. These organizations and individuals creating information objects have all different needs and expectations regarding digital preservation. A musical masterpiece like Beethoven's Symphony No. 9, for example, has to be preserved for many generations to come since it is a part of the cultural heritage. Additionally, it should be stored along with the lyrics, which are based on a poem by Schiller, as well as some good recordings and it should be easily accessible by everyone. On the other hand, a record regarding insurance details of a client has to be kept for just a few decades. An insurance record can most likely be represented in plain text and not some musical notation and contains no binary data like musical recordings. Furthermore, the access to the data must be restricted to those people who have the privileges to view it. After some years have passed and the insurance company is no longer obligated to preserve the records preservation of the insurance record becomes fairly unimportant. It might still be used to gather knowledge about market development or might be interesting for some future generation to analyse past economic systems, but the initial reason for archiving, the obligation by law, is gone. Then again, an arcade game like Pong should be preserved too since it is one of the first video games and contributed notably to the popularity of video games. However, it is an application and requires emulation, reimplementation or some other preservation strategy to be preserved.

These examples show that information objects are quite different and their preservation has to meet diverse requirements. Some consist of plain text, some of binary data, some require restricted access, some should be accessible by everyone, some have to be kept according to laws, some are just of personal value, some should be known by future generations, some might be only interesting in the next 50 years, some should be preserved as long as possible, etc.. Thus, information objects can be quite different, which requires different digital preservation approaches to meet the different preservation needs in a satisfactory way. Therefore, it is not feasible to define a DSL for the domain digital preservation. A DSL which is designed to aid in the preservation of plain text (insurance records) like XML is not useful for the preservation of applications (Pong), which would be better supported by a DSL for emulator development, or a DSL for program specification. As stated above, DSLs are designed to provide expressiveness and ease of use for a domain by abandoning generality and having a narrow focus. Accordingly, a DSL suitable for supporting all the different tasks and requirements of digital preservation would not fit the definition of domain-specific languages. Previously, we illustrated that DSLs can support digital preservation tasks usefully, but creating a single DSL for digital preservation is not a good solution. Digital preservation is better supported by a collection of DSLs tailored for specific tasks.

4 Supporting Knowledge-Management

In [12], Walsh and Ungson established through a literature review that too much focus on preserved knowledge can lead to disregard for present circumstances, which could worsen the performance of an organisation. On the other hand, they determined that preserved knowledge can improve the performance of an organisation because it can improve the understanding of a situation and ease handling of a situation by retaining previously successful or unsuccessful courses of action in similar situations. Hence, given that present circumstances are not disregarded, it is useful to preserve knowledge in order to improve the performance of an organisation. In [1], Alavi and Leidner present different perspectives on knowledge that lead to different implications for knowledge-management. However, whether it is argued that knowledge exists only in the mind of an individual, can be codified and stored, is a process of applying know-how, or that knowledge is a capability to influence future action, we posit that to some degree it is useful to store information/knowledge. For the sake of clarity, we will adopt the postulate of Alavi and Leidner that "information is converted to knowledge once it is processed in the mind of individuals and knowledge becomes information once it is articulated and presented in the form of text, graphics, words, or other symbolic forms" [1]. Thus, from this perspective preservation of information leads to the preservation of knowledge because the preserved information can be converted to knowledge.

Combination of both arguments implicates that it is useful to preserve information so that the information can be converted to knowledge if necessary. Consequently, digital preservation, which, as stated above, entails managing and preserving information to ensure easy discovery and retrieval of the information in present and future situations [4], can support knowledge-management. Employment of digital preservation techniques in knowledge-management is beneficial to ensure that knowledge based on digital information remains accessible over time. Efforts put into the compilation of knowledge should not be rendered useless just because digital information cannot be accessed, read, or interpreted anymore. Furthermore, in addition to their utility for digital preservation by enhancing the emulation strategy or facilitating format standardisation, DSLs provide direct support for knowledge-management. Through the domain context, which increases expressiveness [5], and the capabilities for concise expression, good comprehensibility and avoidance of clerical mistakes [13], DSLs provide direct support for knowledge-management by, for example, avoiding ambiguity or easing conversion of knowledge to information and processing of information to knowledge.

In cases where knowledge cannot be converted to digital information or knowledge might be lost during conversion, digital preservation techniques and DSLs are not of much use for knowledge-management. However, for knowledge that can be converted to information in an adequate way, appropriateness and profitableness of digital preservation techniques and utilisation of DSLs should be considered to ensure that knowledge is not lost and can be better renewed from digital information.

5 Conclusion

In conclusion it has to be said that DSLs can be useful for digital preservation and can make digital preservation tasks easier, but the use of a DSL is no magic bullet so that the appropriateness and profitableness of using a DSL should be carefully considered. DSLs are tailored to a specific application domain and provide expressiveness and ease of use by

abandoning generality. They can be designed to fit individual needs or be popular and widely-used (e.g. HTML). DSLs can be implemented as new languages, language extensions, or can be created with COTS products. In contrast to GPLs, DSLs provide a lot of advantages that outbalance the disadvantages of DSLs in many cases. They are human-legible, use domain-specific notations, provide better expressiveness, increase ease of use, productivity, conciseness, and comprehensibility, and enable end-user development. On the other hand, development of DSLs is difficult since language development expertise as well as expert domain knowledge is required, development of training material, language support, standardisation, and maintenance are costly, there might be bad tool support, and developers are usually more familiar with developing using a GPL.

A DSL for emulator development could enhance the emulation strategy by reducing the necessary effort. The DSL XML is useful for digital preservation because it can support preservation tasks by providing format standardisations for information stored in plain text files. XML could be helpful for archiving binary data (e.g. video, audio) by providing meta-data how the bitstreams should be interpreted, but it is most useful for archiving plain text data. Although digital preservation can be considered a domain there is no DSL supporting digital preservation in general since digital preservation entails too many diverse tasks and fields of action.

In summary, this shows that DSLs are an object for consideration in digital preservation projects. Accordingly, their application can be rewarding in knowledge-management to improve preservation of knowledge encapsulated in digital information, to keep that knowledge retrievable, and to enhance expressiveness, conciseness and comprehensibility of the digital information. DSLs can be useful as long as their application is considered carefully so that they fit the tasks to be done and the objectives to be achieved. DSLs should just not be used blindly. As long as they are used in the right way they can unfold huge potential for the improvement of digital preservation.

6 Bibliography

[1] Alavi, M.; Leidner, D. (2001): Review: Knowledge Management and Knowledge Management Systems: Conceptual Foundations and Research Issues. In: MIS Quarterly 25(1):107-136 http://knol.google.com/k/knowledge-management-back-to-basic-principles.

[2] Landin, P. (1966): The next 700 Programming Languages. In: Communications of the ACM 9(3):157-166.

[3] Lee, K. et al. (2002): The State of the Art and Practice in Digital Preservation. In: Journal of Research of the National Institute of Standards and Technology 107(1):93-106.

[4] Madnick, S. et al. (2009): Overview and Framework for Data and Information Quality Research. In: Journal of Data and Information Quality 1(1):1-22.

[5] Mernik, M.; Heering, J.; Sloane, A. (2005): When and how to develop domain-specific languages. In: ACM Computing Surveys 37(4):316-344.

[6] Rothenberg, J. (2002): Preservation of the Times. In: The Information Management Journal 36(2):38-43.

[7] Rost, R. et al. (2009): OpenGL Shading Language. Addison Wesley.

[8] Spinellis, D. (2001): Notable design patterns for domain specific languages. In: Journal of Systems and Software 56(1):91-99.

[9] Steyer, R. (2005): XML mit Java. Professionell einsteigen. n.p.

[10] Slats, J.; Verdegem, R. (2004): Practical experiences of the Dutch Digital Preservation Testbed. http://www.archief.nl/sites/default/files/docs/kennisbank/article_in_vine_2004 .pdf; retrieved 09.01.2011.

[11] W3C (2008): Extensible Markup Language (XML) 1.0. http://www.w3.org/TR/2008/REC-xml-20081126; retrieved 06.01.2011.

[12] Walsh, J.; Ungson, G. (1991): Organizational Memory. In: The Academy of Management Review 16(1):57-91.

[13] Wile, D. (2001): Supporting the DSL Spectrum. In: Journal of Computing and Information Technology 9(4):263-287.

[14] Wile, D.; Ramming, J. (1999): Guest Editorial: Introduction to the Special Section. In: IEEE Transactions on Software Engineering 25(3):289-290.

[15] Young, R (2008): Knowledge Management – Back to Basic Principles. http://knol.google.com/k/knowledge-management-back-to-basic-principles; retrieved 19.08.2011